AJAY KADIYALA - Data Engineer

Follow me Here:

LinkedIn:
https://www.linkedin.com/in/ajay026/

Data Geeks Community:

https://lnkd.in/gU5NkCqi

PYTHON INTERVIEW KIT FOR DATA ENGINEERS

Python is essential for data engineers because it's incredibly versatile and powerful for handling data. From building data pipelines and processing large datasets to automating workflows, Python offers libraries like Pandas, NumPy, and PySpark that make data manipulation easy and efficient. It’s also great for integrating with databases, working with APIs, and managing cloud services. Plus, Python's simplicity means you can write clean, readable code that scales well, which is crucial when you're dealing with big data and complex systems. Overall, it's a go-to tool for solving real-world data challenges efficiently.

This document contains 100 medium to advanced Python questions designed for data engineering interviews. It also contains the resources to learn from, best practices. One stop solution for your data interviews. It covers key areas like Python basics, data structures, algorithms, libraries such as Pandas and NumPy, file handling, parallel processing, and scenario-based questions, all aimed at helping you prepare for technical interviews.

Table of Contents

7. Working with APIs

8. Error Handling

9. Scenario-Based Questions

10. Bonus: Best Practices in Python for Data Engineering

11. FREE Resources

Python Basics for Data Engineering

Question: Explain how Python's list comprehensions work. Write a list comprehension to filter and square only the even numbers from a given list.

Solution:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_squares = [x ** 2 for x in numbers if x % 2 == 0]
```

Explanation:

List comprehensions are a compact way to process elements in a list. Here, we iterate through 'numbers' and square only the even ones.

Data Structures & Algorithms

Question: Write a Python function to reverse a linked list in-place, assuming you have a singly linked list.

Solution:

```
class Node:
    def __init__(self, value):
        self.value = value
```

```
        self.next = None

def reverse_linked_list(head):
    prev = None
    current = head
    while current is not None:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    return prev
```

Explanation:

This function iteratively reverses the linked list by adjusting the 'next' pointers of the nodes. We maintain references to the current node, previous node, and next node during the iteration.

Pandas Library

Question: Using Pandas, write a function that reads a CSV file and returns the top 5 rows where a particular column's values are greater than a given threshold.

Solution:

```
import pandas as pd
def filter_csv(file_path, column_name, threshold):
    df = pd.read_csv(file_path)
    return df[df[column_name] > threshold].head(5)
```

Explanation:

This function reads a CSV into a DataFrame, filters rows where a specified column exceeds the threshold, and returns the first 5 rows.

File Handling

Question: Write a Python function to read a large CSV file in chunks and process each chunk to calculate the average of a column.

Solution:

```
import pandas as pd
def process_large_csv(file_path, chunk_size=1000):
    total_sum = 0
    total_count = 0
    for chunk in pd.read_csv(file_path, chunksize=chunk_size):
        total_sum += chunk['column_name'].sum()
        total_count += len(chunk)
    return total_sum / total_count
```

Explanation:

By reading the CSV file in chunks using Pandas, we avoid loading the entire file into memory. Each chunk's sum and count are computed, and the average is returned at the end.

Parallel Processing

Question: Using Python's multiprocessing library, write a function to process a list of tasks in parallel and return the results.

Solution:

```
from multiprocessing import Pool
def process_task(task):
    # Process the task
    return task ** 2  # Example: Square the number

def parallel_processing(tasks):
    with Pool() as pool:
        results = pool.map(process_task, tasks)
    return results
```

Explanation:

This function uses Python's multiprocessing.Pool to process a list of tasks in parallel. Each task is squared in this example, but the function can be customized for different workloads.

Working with APIs

Question: Write a Python function to fetch data from a REST API using the requests library and handle pagination until all data is retrieved.

Solution:

```
import requests
def fetch_api_data(base_url):
    results = []
    while base_url:
        response = requests.get(base_url)
        data = response.json()
        results.extend(data['results'])
```

```
    base_url = data.get('next')  # Update to the next page
  return results
```

Explanation:

This function makes repeated API requests, following pagination links until all data is retrieved. Each response's results are added to the final output.

Scenario-Based: Optimizing a Data Pipeline

Question: Given a Python-based ETL pipeline that processes daily logs, how would you optimize it for performance and scalability when the data size grows?

Solution:

1. Use multiprocessing to parallelize the log processing steps.
2. Read the logs in chunks to avoid memory overflow.
3. Use efficient data structures like Pandas DataFrame or Dask for large data.
4. If using a cloud platform, consider using services like AWS Lambda or Apache Spark to distribute the load.
5. Profile the code to identify bottlenecks and optimize specific functions.

Explanation:

To handle growing data size, multiprocessing, chunking, and distributed systems (like Spark) are key strategies for performance optimization. Profiling helps locate bottlenecks for targeted improvements.

Error Handling

Question: Write a Python function to handle both file reading errors and division by zero errors in a single try-except block.

Solution:

```
def safe_file_and_division(file_path, divisor):
    try:
        with open(file_path, 'r') as f:
            data = f.read()
        result = 100 / divisor
    except FileNotFoundError:
        return 'File not found!'
    except ZeroDivisionError:
        return 'Cannot divide by zero!'
    return result
```

Explanation:

This function catches multiple exceptions: one for file-related errors and another for division by zero. The use of try-except blocks makes the code robust to these common issues.

Scenario-Based: Data Transformation

Question: You're working with a large dataset in CSV format that contains millions of rows. The dataset contains missing values, and you need to clean it by filling the missing values with the median of each column. How would you do this using Pandas?

Solution:

```
import pandas as pd
def clean_dataset(file_path):
```

```
    df = pd.read_csv(file_path)
    for column in df.columns:
        if df[column].isnull().sum() > 0:
            median_value = df[column].median()
            df[column].fillna(median_value, inplace=True)
    return df
```

Explanation:

This function reads the dataset into a Pandas DataFrame and iterates through each column to fill missing values with the median. This approach ensures the dataset is clean before further analysis.

Scenario-Based: Memory-Efficient Data Processing

Question: How would you process a huge dataset that cannot fit into memory using Python, ensuring efficient memory usage?

Solution:

```
import pandas as pd
def process_large_dataset_in_chunks(file_path,
chunk_size=10000):
    for chunk in pd.read_csv(file_path, chunksize=chunk_size):
        # Perform processing on each chunk
        print(chunk.describe())
```

Example: Print summary stats for each chunk

Explanation:

Using the `chunksize` parameter in Pandas allows for processing large datasets in manageable chunks without loading the entire file into memory. This approach ensures memory efficiency.

Scenario-Based: Working with APIs and Real-Time Data

Question: You need to write a Python program to fetch real-time weather data every hour and save it to a file. How would you design this using the requests and time libraries?

Solution:

```
import requests
import time

def fetch_weather_data(api_url, save_path):
    while True:
        response = requests.get(api_url)
        if response.status_code == 200:
            with open(save_path, 'a') as f:
                f.write(response.text + '\n')
        time.sleep(3600)  # Sleep for one hour
```

Explanation:

This function fetches weather data every hour and appends it to a file. The `time.sleep()` function ensures the program runs continuously at hourly intervals.

Data Structures: Efficient Dictionary Usage

Question: You are tasked with counting the frequency of each element in a list. Write a Python function using a dictionary to achieve this efficiently.

Solution:

```
def count_frequencies(data):
    frequency_dict = {}
    for item in data:
        if item in frequency_dict:
            frequency_dict[item] += 1
        else:
            frequency_dict[item] = 1
    return frequency_dict
```

Explanation:

Using a dictionary is efficient for counting frequencies. This solution iterates through the list, updating the dictionary with the count of each element.

Working with Date and Time

Question: Write a Python function that takes two date strings (in the format 'YYYY-MM-DD') and returns the number of days between them.

Solution:

```
from datetime import datetime
def days_between_dates(date1, date2):
    date_format = '%Y-%m-%d'
    d1 = datetime.strptime(date1, date_format)
    d2 = datetime.strptime(date2, date_format)
    return abs((d2 - d1).days)
```

Explanation:

This function parses the date strings into `datetime` objects and calculates the absolute difference in days between the two dates.

Data Processing: Parsing JSON Files

Question: You have a large JSON file containing nested data structures. Write a Python function to extract specific fields from the JSON and save them into a CSV file using Pandas.

Solution:

```
import pandas as pd
import json

def parse_json_to_csv(json_file, csv_file):
    with open(json_file, 'r') as f:
        data = json.load(f)
    # Extract specific fields (example: name, age)
    df = pd.DataFrame(data['people'], columns=['name', 'age'])
    df.to_csv(csv_file, index=False)
```

Explanation:

This function loads a JSON file, extracts specific fields from nested data, and writes the result to a CSV file using Pandas.

File Handling: Working with Multiple Files

Question: You have several text files in a directory, each containing data on a specific topic. Write a Python script to read all these files, combine their content into a single file, and remove duplicates.

Solution:

```
import os

def combine_files(directory, output_file):
    content_set = set()
    for file_name in os.listdir(directory):
        if file_name.endswith('.txt'):
            with open(os.path.join(directory, file_name), 'r') as f:
                content = f.read()
                content_set.update(content.splitlines())
    with open(output_file, 'w') as f:
        f.write('\n'.join(content_set))
```

Explanation:

This script reads all text files in the given directory, combines their content, removes duplicates by using a set, and writes the combined content to a new file.

Advanced: Using Generators for Memory Efficiency

Question: How would you implement a generator in Python to yield large data chunks from a file instead of loading the entire file into memory at once?

Solution:

```
def read_large_file_in_chunks(file_path, chunk_size=1024):
    with open(file_path, 'r') as f:
        while True:
            data = f.read(chunk_size)
            if not data:
```

```
            break
        yield data
```

Explanation:

This generator reads a large file in chunks, yielding each chunk at a time. This ensures that only a portion of the file is loaded into memory, making the function memory efficient.

Error Handling: Custom Exception Classes

Question: Write a custom exception class in Python for handling specific database connection errors. Use it in a function that simulates connecting to a database.

Solution:

```
class DatabaseConnectionError(Exception):
    pass

def connect_to_database(db_url):
    if db_url != 'valid_url':
        raise DatabaseConnectionError('Failed to connect to the database')
```

Explanation:

This code defines a custom exception class `DatabaseConnectionError` and raises it when a simulated database connection fails. Custom exceptions make error handling more specific.

Parallel Processing: Multiprocessing with Queues

Question: Using the multiprocessing library, write a Python program that spawns multiple processes to process items in a shared queue.

Solution:

```
from multiprocessing import Process, Queue

def worker(q):
    while not q.empty():
        item = q.get()
        print(f'Processing item {item}')

def parallel_queue_processing(data):
    q = Queue()
    for item in data:
        q.put(item)
    processes = [Process(target=worker, args=(q,)) for _ in range(4)]
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

Explanation:

This program creates multiple processes to process items in a shared queue. The multiprocessing. Queue is used to share tasks between processes.

Optimization: Profiling Python Code

Question: You need to optimize a slow-running Python function. How would you profile the function to identify the bottleneck?

Solution:

```
import cProfile

def slow_function():
    total = 0
    for i in range(1000000):
        total += i
    return total

# Profile the function
cProfile.run('slow_function()')
```

Explanation:

The `cProfile` module allows you to profile the performance of Python code and identify bottlenecks. This example profiles a function that sums a large number of integers.

Scenario-Based: Optimizing API Calls

Question: You're tasked with making multiple API calls to retrieve data from an external service. However, the service has a rate limit of 100 requests per minute. How would you optimize your Python script to respect this limit while retrieving the data efficiently?

Solution:

```
import requests
import time

def make_api_calls(urls):
    results = []
    for index, url in enumerate(urls):
        if index > 0 and index % 100 == 0:
            time.sleep(60)  # Respect the 100 requests per minute limit
        response = requests.get(url)
        if response.status_code == 200:
            results.append(response.json())
    return results
```

Explanation:

This script makes API calls while respecting the rate limit by pausing execution every 100 requests. Using time.sleep() ensures that the limit is not exceeded.

File Handling: Reading and Writing Binary Files

Question: Write a Python function that reads a binary file and writes its contents to a new file, reversing the byte order in the process.

Solution:

```
def reverse_bytes(input_file, output_file):
    with open(input_file, 'rb') as f:
        data = f.read()
    reversed_data = data[::-1]
```

```
    with open(output_file, 'wb') as f:
        f.write(reversed_data)
```

Explanation:

This function reads the binary content of a file, reverses the byte order, and writes the reversed data to a new file.

Parallel Processing: Using Threading for I/O Bound Tasks

Question: Write a Python program using threading to download multiple files simultaneously. The files should be saved to a directory after downloading.

Solution:

```
import threading
import requests

def download_file(url, save_path):
    response = requests.get(url)
    with open(save_path, 'wb') as f:
        f.write(response.content)

def download_files(urls, save_dir):
    threads = []
    for url in urls:
        file_name = url.split('/')[-1]
        save_path = f'{save_dir}/{file_name}'
        thread = threading.Thread(target=download_file, args=(url,
save_path))
        thread.start()
```

```
        threads.append(thread)
    for thread in threads:
        thread.join()
```

Explanation:

This program uses threading to download multiple files in parallel. Each thread is responsible for downloading and saving a file to a specified directory.

Advanced: Implementing a Cache with LRU Cache

Question: Explain how Python's functools.lru_cache works. Write a Python function that uses LRU caching to optimize a recursive Fibonacci function.

Solution:

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Explanation:

The LRU (Least Recently Used) cache stores up to 'maxsize' recent function calls to optimize expensive function calls like recursive Fibonacci calculations.

Working with Dates and Times: Time Zone Conversions

Question: Write a Python function that converts a given UTC timestamp to a specific time zone using the pytz library.

Solution:

```
from datetime import datetime
import pytz

def convert_utc_to_timezone(utc_time_str, timezone_str):
    utc = pytz.utc
    local_timezone = pytz.timezone(timezone_str)
    utc_time = utc.localize(datetime.strptime(utc_time_str, '%Y-%m-%d %H:%M:%S'))
    local_time = utc_time.astimezone(local_timezone)
    return local_time
```

Explanation:

This function uses the `pytz` library to convert a UTC timestamp to a specific time zone, ensuring accurate time zone conversions.

Scenario-Based: Parsing Large JSON Data

Question: You have a large JSON file with millions of records. How would you efficiently parse it in Python without loading the entire file into memory?

Solution:

```
import json

def parse_large_json(file_path):
```

```
  with open(file_path, 'r') as f:
    for line in f:
      record = json.loads(line)
      # Process each record
      print(record)
```

Explanation:

This function reads a large JSON file line by line, parsing each record without loading the entire file into memory. This approach ensures memory efficiency.

Optimization: Using NumPy for Fast Calculations

Question: You are tasked with calculating the dot product of two large matrices. How would you do this efficiently in Python using NumPy?

Solution:

```
import numpy as np

def calculate_dot_product(matrix_a, matrix_b):
  return np.dot(matrix_a, matrix_b)
```

Explanation:

NumPy's `np.dot` function is optimized for fast matrix operations, making it ideal for large-scale mathematical computations like dot products.

Advanced: Using Decorators for Function Timing

Question: Write a Python decorator that times how long a function takes to execute and logs the execution time.

Solution:

```
import time

def time_function(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f'Function {func.__name__} took {end_time -
start_time:.2f} seconds to execute')
        return result
    return wrapper

@time_function
def example_function():
    time.sleep(2)

# example_function() will log its execution time
```

Explanation:

The `time_function` decorator wraps around any function and calculates the time it takes to execute. The result is printed after the function completes.

Scenario-Based: Processing Data in Real-Time

Question: You are asked to build a real-time data processing system using Python to ingest and process live data streams. What libraries would you use, and how would you design the system?

Solution:

1. Use a message broker like Kafka for data ingestion.
2. Write a consumer using the kafka-python library to consume messages in real-time.
3. Use Pandas or Dask for real-time data processing and aggregation.
4. Store processed results in a database for later analysis.

Explanation:

This system uses Kafka to handle real-time data streams, with kafka-python for message consumption. Pandas or Dask is used for processing, and a database is used for storage.

Advanced: Writing Tests with pytest

Question: Write a unit test using pytest to test a function that adds two numbers.

Solution:

```
def add_numbers(a, b):
    return a + b

def test_add_numbers():
    assert add_numbers(2, 3) == 5
    assert add_numbers(-1, 1) == 0

# To run this test, save it in a file and run `pytest <filename>.py`
```

Explanation:

Using `pytest`, you can write unit tests that check the correctness of a function. In this case, the `test_add_numbers` function ensures the `add_numbers` function behaves as expected.

File I/O: Handling Compressed Files

Question: Write a Python function to read a GZIP compressed CSV file and return its contents as a Pandas DataFrame.

Solution:

```
import pandas as pd
import gzip

def read_gzip_csv(file_path):
    with gzip.open(file_path, 'rt') as f:
        df = pd.read_csv(f)
    return df
```

Explanation:

This function opens a GZIP compressed file using `gzip.open` and reads its content into a Pandas DataFrame. The 'rt' mode is used for reading text files.

Advanced: Memory Profiling in Python

Question: You need to profile a Python script to find memory bottlenecks. Write a Python function that uses the memory_profiler library to track memory usage of a specific function.

Solution:

```
from memory_profiler import profile

@profile
def memory_intensive_function():
    a = [i for i in range(1000000)]  # Example of memory-intensive operation
    return a

# To run this, use the command: mprof run <script_name.py>
# Then view the memory profile: mprof plot
```

Explanation:

The `memory_profiler` library tracks the memory usage of the decorated function. The `@profile` decorator shows the memory consumption of each line.

Error Handling: Logging Exceptions to a File

Question: Write a Python function that logs exceptions to a file using the logging module whenever an error occurs.

Solution:

```
import logging

def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        logging.basicConfig(filename='error.log', level=logging.ERROR)
```

```
    logging.error(f'Error occurred: {e}')
    return 'Error: Cannot divide by zero!'
```

Explanation:

This function uses the `logging` module to log exceptions to a file named 'error.log'. When an error occurs, the exception is logged with the ERROR level.

Scenario-Based: Data Cleaning with Pandas

Question: Given a dataset with columns containing strings and NaN values, write a Python function using Pandas to clean the data by removing leading/trailing spaces and filling NaNs with a default value.

Solution:

```
import pandas as pd

def clean_data(df):
   df = df.apply(lambda x: x.str.strip() if x.dtype == 'object' else x)
# Remove spaces
   df.fillna('N/A', inplace=True)  # Fill NaNs with 'N/A'
   return df
```

Explanation:

This function removes leading/trailing spaces in string columns and fills NaN values with 'N/A' using Pandas' `apply()` and `fillna()` methods.

File Handling: Processing Large Files Line by Line

Question: Write a Python function to process a large text file line by line and count the number of lines containing a specific word.

Solution:

```
def count_lines_with_word(file_path, word):
    count = 0
    with open(file_path, 'r') as f:
        for line in f:
            if word in line:
                count += 1
    return count
```

Explanation:

This function opens a large file, processes it line by line to avoid memory issues, and counts how many lines contain the specified word.

Scenario-Based: Managing Rate-Limited API Calls

Question: You need to retrieve data from a rate-limited API (e.g., 50 requests per minute). Write a Python function to ensure that you do not exceed this limit.

Solution:

```
import requests
import time

def fetch_data_with_rate_limit(urls):
    results = []
```

```
    for index, url in enumerate(urls):
        if index > 0 and index % 50 == 0:
            time.sleep(60)  # Pause for 60 seconds after 50 requests
        response = requests.get(url)
        if response.status_code == 200:
            results.append(response.json())
    return results
```

Explanation:

This function uses a counter to track the number of requests made. After every 50 requests, it pauses for 60 seconds to respect the rate limit.

Concurrency: Using asyncio for I/O-bound Tasks

Question: Write a Python function that uses the asyncio library to make multiple API calls concurrently.

Solution:

```
import asyncio
import aiohttp

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.json()

async def fetch_all(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(session, url) for url in urls]
        return await asyncio.gather(*tasks)
```

```
# asyncio.run(fetch_all(url_list))
```

Explanation:

This function uses `aiohttp` for asynchronous HTTP requests and `asyncio.gather()` to run multiple API calls concurrently.

Advanced: Data Transformation with Dask

Question: Write a Python function using Dask to perform a large-scale groupby operation on a dataset that is too large to fit into memory.

Solution:

```
import dask.dataframe as dd

def large_scale_groupby(file_path, column):
    df = dd.read_csv(file_path)
    result = df.groupby(column).agg({'value': 'sum'}).compute()
    return result
```

Explanation:

Dask allows you to perform out-of-memory operations on large datasets. The `.compute()` method is used to execute the operation and return the result.

Scenario-Based: Working with Time Series Data

Question: You have a time series dataset containing stock prices. Write a Python function to calculate the rolling average of stock prices over a 7-day window.

Solution:

```
import pandas as pd

def calculate_rolling_average(df):
    df['7_day_avg'] = df['stock_price'].rolling(window=7).mean()
    return df
```

Explanation:

This function uses Pandas' `rolling()` method to calculate the rolling average of stock prices over a 7-day window.

Error Handling: Retry Failed API Calls

Question: Write a Python function that retries an API call up to 3 times if it fails, using the requests library.

Solution:

```
import requests

def fetch_with_retries(url, retries=3):
    for _ in range(retries):
        try:
            response = requests.get(url)
            if response.status_code == 200:
                return response.json()
        except requests.RequestException:
            continue
    return 'Failed after 3 retries'
```

Explanation:

This function retries an API call up to 3 times in case of failure. It uses a loop to make the request and only returns data if the response status is 200.

Advanced: Using SQLAlchemy for Database Operations

Question: Write a Python function that uses SQLAlchemy to query a database for records where the value of a specific column exceeds a threshold.

Solution:

```
from sqlalchemy import create_engine, Table, MetaData

def query_database(threshold):
    engine = create_engine('sqlite:///example.db')
    conn = engine.connect()
    metadata = MetaData(bind=engine)
    table = Table('data', metadata, autoload=True)
    query = table.select().where(table.c.value > threshold)
    result = conn.execute(query)
    return result.fetchall()
```

Explanation:

SQLAlchemy provides an ORM-like interface for querying databases. This function connects to a SQLite database, defines the table schema, and runs a query with a threshold.

Scenario-Based: Data Deduplication with Pandas

Question: You have a dataset containing duplicate rows. Write a Python function using Pandas to remove duplicates and return the cleaned dataset.

Solution:

```
import pandas as pd

def remove_duplicates(df):
    return df.drop_duplicates()
```

Explanation:

The Pandas `drop_duplicates()` method removes duplicate rows from a dataset, returning a cleaned DataFrame.

Performance Optimization: Vectorization with NumPy

Question: Write a Python function using NumPy to replace a loop-based operation with a vectorized operation for performance improvement.

Solution:

```
import numpy as np

def vectorized_operation(arr):
    return np.sqrt(arr)  # Vectorized square root operation

# Replaces:
# result = [math.sqrt(x) for x in arr]
```

Explanation:

NumPy's vectorized operations (such as `np.sqrt()`) are much faster than traditional Python loops, making them ideal for performance optimization.

Scenario-Based: Parallel Processing with Concurrent Futures

Question: Write a Python function using the concurrent.futures module to run multiple I/O-bound tasks in parallel and return the results.

Solution:

```
import concurrent.futures
import requests

def fetch_url(url):
    response = requests.get(url)
    return response.text

def parallel_url_fetch(urls):
    results = []
    with concurrent.futures.ThreadPoolExecutor() as executor:
        future_to_url = {executor.submit(fetch_url, url): url for url in
urls}
        for future in concurrent.futures.as_completed(future_to_url):
            results.append(future.result())
    return results
```

Explanation:

This function uses `concurrent.futures.ThreadPoolExecutor` to run multiple I/O-bound tasks, such as fetching URLs in parallel, and return the results as they complete.

Error Handling: Using Custom Exceptions

Question: Write a custom exception class in Python and demonstrate how to raise and catch it in a function.

Solution:

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message

def risky_operation(value):
    if value < 0:
        raise CustomError('Negative value error!')
    return value ** 2

try:
    risky_operation(-5)
except CustomError as e:
    print(f'Caught an error: {e.message}')
```

Explanation:

This code defines a custom exception class `CustomError` and demonstrates how to raise and catch it during the execution of a function.

Data Wrangling: Merging Multiple DataFrames

Question: You have two DataFrames containing employee and salary data. Write a Python function using Pandas to merge them on the employee ID and return the merged result.

Solution:

```
import pandas as pd

def merge_dataframes(df1, df2):
    return pd.merge(df1, df2, on='employee_id')
```

Explanation:

This function merges two DataFrames using Pandas' `merge()` method based on the `employee_id` column, returning the combined result.

Scenario-Based: Processing Large XML Files

Question: Write a Python function using the xml.etree.ElementTree library to parse a large XML file and extract specific elements without loading the entire file into memory.

Solution:

```
import xml.etree.ElementTree as ET

def parse_large_xml(file_path, tag_name):
    context = ET.iterparse(file_path, events=('end',))
    for event, elem in context:
        if elem.tag == tag_name:
            yield elem.text
        elem.clear()  # Free memory by clearing processed elements
```

Explanation:

This function uses `iterparse()` to parse an XML file iteratively, allowing it to extract specific elements without loading the entire file into memory. The `clear()` method is used to free up memory.

Concurrency: Implementing Asynchronous Database Queries

Question: Write a Python function using the asyncpg library to query a PostgreSQL database asynchronously.

Solution:

```
import asyncpg
import asyncio

async def query_database():
    conn = await asyncpg.connect(user='user',
password='password', database='db', host='localhost')
    rows = await conn.fetch('SELECT * FROM table_name')
    await conn.close()
    return rows

# asyncio.run(query_database())
```

Explanation:

This function uses the `asyncpg` library to execute asynchronous queries to a PostgreSQL database, ensuring efficient use of resources by not blocking the main thread.

Advanced: Implementing a REST API with FastAPI

Question: Write a Python function to implement a basic REST API endpoint using the FastAPI framework, which returns a list of items.

Solution:

```
from fastapi import FastAPI

app = FastAPI()

@app.get('/items/')
def get_items():
    return [{'item_id': 1, 'name': 'Item A'}, {'item_id': 2, 'name': 'Item B'}]

# Run the FastAPI app using: uvicorn script_name:app --reload
```

Explanation:

This function uses the FastAPI framework to create a simple REST API endpoint that returns a list of items. FastAPI is known for its speed and asynchronous capabilities.

File Handling: Writing Data to Excel Files

Question: Write a Python function using the openpyxl library to write data to an Excel file.

Solution:

```
from openpyxl import Workbook

def write_to_excel(data, file_path):
    wb = Workbook()
    ws = wb.active
    for row in data:
        ws.append(row)
    wb.save(file_path)
```

Explanation:

This function creates an Excel workbook using the `openpyxl` library, writes the provided data to it, and saves the file to the specified path.

Scenario-Based: Processing Real-Time Data with Kafka

Question: Write a Python function using the kafka-python library to consume real-time messages from a Kafka topic.

Solution:

```
from kafka import KafkaConsumer

def consume_messages(topic):
    consumer = KafkaConsumer(topic,
bootstrap_servers='localhost:9092')
    for message in consumer:
        print(f'Received message: {message.value}')
```

Explanation:

This function uses the `kafka-python` library to consume real-time messages from a specified Kafka topic and prints the received messages.

Performance Optimization: Memoization with functools.lru_cache

Question: Write a Python function that uses the lru_cache decorator from functools to optimize a recursive factorial function.

Solution:

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Explanation:

The `lru_cache` decorator from the `functools` module caches the results of expensive function calls, significantly improving the performance of recursive functions like factorial.

Data Processing: Filtering Data with PySpark

Question: Write a Python function using PySpark to filter rows from a DataFrame where the value in a specific column exceeds a threshold.

Solution:

```
from pyspark.sql import SparkSession

def filter_spark_dataframe(file_path, column, threshold):
    spark =
SparkSession.builder.appName('DataProcessing').getOrCreate()
    df = spark.read.csv(file_path, header=True, inferSchema=True)
    filtered_df = df.filter(df[column] > threshold)
    return filtered_df
```

Explanation:

This function uses PySpark to filter rows from a DataFrame where the value in a specified column exceeds the given threshold, allowing for scalable data processing.

My Interview Experience:

What is lazy evaluation in Python, and how can it improve the efficiency of data processing pipelines?

Answer: Lazy evaluation defers computation until the results are actually needed, which helps save memory and improve performance. For example, generators in Python only produce one item at a time, conserving memory when processing large data streams.

Solution

```
def process_large_data(data):
    for item in data:
        yield process(item)
```

This method, process_large_data, doesn't load everything into memory but generates each result as needed.

How would you handle very large datasets that do not fit into memory in Python?

Answer: For large datasets, you can use chunking or libraries like Dask. Chunking involves processing data in smaller pieces, such as reading large CSV files in batches.

Solution:

```
import pandas as pd

for chunk in pd.read_csv('large_file.csv', chunksize=10000):

    process(chunk)
```

Dask also enables handling big data in Python without loading it entirely into memory.

3. Can you optimize this Python function?

Answer: Refactor to use a generator, reducing memory consumption.

Solution:

```
def process_file(filename):

    with open(filename, 'r') as file:

        return (process_line(line) for line in file)
```

How would you implement a simple custom cache in Python to improve function performance?

Answer: We can use a dictionary or functools.lru_cache to cache results.

Solution:

```
from functools import lru_cache
```

```
@lru_cache(maxsize=1000)
def expensive_function(x):
    # Function logic here
```

This lru_cache will store the last 1,000 results, improving performance for frequently used values.

How would you troubleshoot memory leaks in a long-running Python ETL process?

Answer: Use tools like tracemalloc or memory_profiler to track memory usage. These tools let you pinpoint where memory usage spikes and identify issues, such as unnecessary variables not being released.

Solution:

```
import tracemalloc

tracemalloc.start()
# ETL process
print(tracemalloc.get_traced_memory())
tracemalloc.stop()
```

Explain the difference between map, filter, and reduce in Python, and when you might use each in data processing.

Answer:

- map: Applies a function to each item in an iterable.
- filter: Filters items based on a condition.
- reduce: Reduces an iterable to a single value using a binary function.

For large datasets, map and filter work well when used with generators.

7. How would you implement data validation in a data ingestion pipeline in Python?

Answer: Use a data validation library like pandera or write custom checks. Ensure values meet the required schema and handle missing values.

Solution:

```
import pandera as pa


schema = pa.DataFrameSchema({
    "column1": pa.Column(int, nullable=False),
    "column2": pa.Column(str, nullable=True)
})
validated_data = schema.validate(data)
```

Explain the benefits and drawbacks of using a framework like Apache Spark in Python over traditional Python libraries for processing large datasets.

Answer: Spark is distributed, so it's more scalable and faster for massive datasets. However, it can be complex to set up and might be overkill for smaller jobs where Pandas or Dask could suffice.

9. What are generators in Python, and how do they improve memory efficiency in data pipelines?

Answer: Generators only compute values as needed, conserving memory. They're useful for processing large files or streams without loading them entirely.

Solution:

```
def line_reader(filename):
    with open(filename) as f:
        for line in f:
            yield line
```

10. How would you handle parallel data processing in Python?

Answer: Use multiprocessing or libraries like Dask. For example, with multiprocessing, each worker processes a chunk of data in parallel.

Solution:

```
from multiprocessing import Pool


with Pool(5) as p:
    results = p.map(process, data_chunks)
```

11. Write a Python function to calculate the moving average of a large dataset streamed in real-time.

Answer: A simple moving average can be calculated with a window.

Solution:

```
def moving_average(stream, window_size=5):
    result = []
    window = []
    for item in stream:
        window.append(item)
        if len(window) > window_size:
            window.pop(0)
        result.append(sum(window) / len(window))
    return result
```

12. Explain how you would implement error handling and logging in a Python data pipeline.

Answer: Use try-except blocks and Python's logging module for structured logging.

Solution:

```
import logging


logging.basicConfig(level=logging.INFO)


def process_data(data):
    try:
        # Process logic
        logging.info("Processing successful")
    except Exception as e:
        logging.error("Error occurred: %s", e)
```

13. How would you use Python to extract and process data from a REST API that has pagination?

Answer: Use a loop to request each page until there are no more pages.

Solution:

```
import requests


url = "https://api.example.com/data"
while url:
    response = requests.get(url)
    data = response.json()
    process(data)
    url = data.get('next_page')
```

14. Describe how you would use Pandas to merge multiple large datasets while handling memory limitations.

Answer: Use chunking and incremental merging. Read data in smaller chunks, merge each chunk, and save intermediate results.

Solution:

```
for chunk in pd.read_csv(file, chunksize=10000):
    # merge logic
```

15. Write a Python function to process a large CSV file, row by row, without loading the entire file into memory.

Answer:

Solution:

```
import csv


def process_large_csv(file_path):
    with open(file_path, 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            process(row)
```

16. What's the difference between deepcopy and copy in Python, and how would you apply it in a data processing context?

Answer: copy.copy creates a shallow copy, while copy.deepcopy creates a full copy, duplicating nested objects.

17. How would you use asyncio in Python to speed up an I/O-bound data processing task?

Answer:

Solution:

```
import asyncio


async def fetch_data(url):
    # async fetch logic here
```

```
async def main():
    urls = ["url1", "url2"]
    await asyncio.gather(*(fetch_data(url) for url in urls))

asyncio.run(main())
```

18. How can you optimize Spark operations in PySpark when handling large datasets?

Answer: Techniques include:

- Using repartition to distribute data evenly
- Filtering early to reduce data size
- Avoiding unnecessary collect calls

Explain how to implement distributed data processing in Python using Dask.

Answer: Dask allows distributed computation across clusters. It uses dataframes, similar to Pandas, but can handle data that doesn't fit in memory by splitting it into chunks processed in parallel.

Solution:

```
import dask.dataframe as dd

df = dd.read_csv('large_file.csv')
```

```
result = df.groupby('column').sum().compute()
```

How would you handle schema evolution in JSON or Avro files during data ingestion?

Answer: Use libraries like fastavro or avro-python3 for Avro data, and manage schema evolution by validating fields, providing defaults, or using backward-compatible schema changes. For JSON, rely on schema validation libraries such as jsonschema.

What are weak references in Python, and when would you use them?

Answer: Weak references allow referencing objects without preventing their garbage collection. They are useful when caching objects without increasing memory usage unnecessarily. The weakref module supports weak references in Python.

How would you manage configuration and secrets securely in Python for a production ETL pipeline?

Answer: Use environment variables or tools like AWS Secrets Manager or Azure Key Vault. Avoid hardcoding sensitive information in code; use os.environ to retrieve variables securely.

Solution:

```
import os

db_password = os.getenv("DB_PASSWORD")
```

Explain the role of @staticmethod and @classmethod in Python, and how each might be useful in data engineering code.

Answer: @staticmethod is used for utility functions that don't require access to the class. @classmethod can alter class variables, such as initializing a class with specific configurations, making it useful for initializing connections in ETL classes.

How would you handle retry logic and exponential backoff in Python?

Answer: Use a retry mechanism with a decorator function or the tenacity library for retries with backoff. Here's an example using tenacity:

Solution:

```
from tenacity import retry, wait_exponential

@retry(wait=wait_exponential(multiplier=1, min=4, max=10))
def fetch_data():
```

How would you deal with skewed data in a distributed data processing job using Python?

Answer: Data skew can be managed by rebalancing data with partitioning techniques, filtering high-frequency keys, or using salting, where random values are added to the partition keys to improve data distribution.

Explain the with statement in Python and why it's preferred for resource management in data engineering.

Answer: The with statement ensures resources are automatically released, such as closing files or database connections, which is essential in data pipelines to avoid memory leaks.

Solution:

```
with open('file.txt', 'r') as file:
    data = file.read()
```

Describe how you would apply window functions in Python to analyze streaming data.

Answer: Window functions allow calculation of metrics over a sliding timeframe. Libraries like Pandas or PySpark can be used for this:

Solution:

```
df['rolling_avg'] = df['column'].rolling(window=3).mean()
```

How would you implement logging levels in Python to differentiate between INFO, DEBUG, ERROR, and WARNING messages?

Answer: Set logging levels to control message verbosity, using logging.basicConfig().

Solution:

```
import logging
logging.basicConfig(level=logging.INFO)
logging.debug("Debug info")
```

```
logging.error("Error occurred")
```

What are the benefits of using pandas groupby and apply functions in a data processing workflow? When would they be inefficient?

Answer: groupby and apply allow easy aggregation and transformation. However, they can be slow with very large dataframes. For performance, consider Dask or PySpark for parallel processing.

How would you design a high-performance, Python-based data pipeline that reads data from multiple sources, transforms it, and writes it to a destination?

Answer: Use a modular approach with source, transform, and sink components, and tools like Apache Airflow for orchestration. Use asynchronous processing for I/O-bound tasks and batch processing for heavy computation.

Explain how yield works in Python and how it helps in processing large datasets.

Answer: yield creates a generator that yields results one at a time, which saves memory by not storing all results at once.

Solution:

```
def large_file_reader(file):
    with open(file) as f:
        for line in f:
            yield line
```

How do you handle null or missing values in Python when performing data transformations?

Answer: Use fillna for filling, dropna to remove nulls, or custom imputation strategies. In Pandas:

Solution:

```
df.fillna(0, inplace=True)
```

Describe how you would monitor and log the performance of a data pipeline in Python.

Answer: Use logging for tracking execution and profiling libraries like cProfile for performance. Collect metrics on duration, memory usage, and error rates.

Explain how you would use the logging library in Python to monitor an ETL process.

Answer: Set up loggers with different levels and output logs to files or external monitoring systems for visibility into ETL processing.

How would you handle errors in a Python data pipeline to ensure reliability?

Answer: Use try-except blocks with specific error handling for retries and alerting. Log errors and send notifications for failures to maintain system reliability.

Describe how you would manage dependencies in a Python-based ETL project.

Answer: Use requirements.txt or poetry for dependency management and pin versions to ensure reproducibility across environments.

What are best practices for naming conventions and code organization in Python for a data engineering project?

Answer: Use descriptive names, follow PEP8 guidelines, separate data processing logic from configuration, and use modular functions to enhance readability and reusability.

How would you design a solution to aggregate real-time data from multiple sources and calculate rolling averages?

Answer: Use a streaming framework like Kafka Streams and a sliding window to aggregate data in real-time. In Python, libraries like PySpark Streaming or Apache Flink are helpful.

Describe how to implement custom serialization in Python to optimize performance when handling large datasets.

Answer: Use pickle for serialization, but consider dill or custom serialization functions for more complex data types. Additionally, protobuf or Avro offer efficient storage.

Explain the role of memoization in data processing and how you would implement it in Python.

Answer: Memoization caches function results to avoid redundant calculations, improving performance in repetitive processing.

Solution:

```
from functools import lru_cache

@lru_cache(maxsize=100)
def heavy_computation(x):
    return x ** 2
```

How do you ensure idempotency in a Python-based ETL job?

Answer: Idempotency ensures that rerunning an ETL job doesn't alter final results. Implement checks or hashes, use upserts, and create checkpointing to handle incremental changes reliably.

What are some common anti-patterns in Python data engineering code, and how would you avoid them?

Answer: Avoid large monolithic functions, excessive memory use without lazy loading, and inconsistent variable names. Stick to clean code principles, and avoid hard-coded configurations.

How would you implement data partitioning in Python to improve the performance of a batch processing job?

Answer: Partition data by keys such as date or ID for efficient reads and processing. In databases, partitioned tables make querying faster by limiting the data scanned.

Write a Python function to parse a nested JSON file and extract specific data fields.

Answer:

Solution:

```
import json

def parse_json(file):
    with open(file) as f:
        data = json.load(f)
    for item in data['items']:
        print(item['desired_field'])
```

How would you optimize Pandas code that processes a large dataset with chained operations?

Answer: Break down chained operations, avoid duplicates, and use vectorized operations where possible. For large data, switch to Dask or PySpark for better performance.

Explain the use of decorators in Python. How could decorators help in monitoring or managing data pipeline functions?

Answer: Decorators add functionality to functions without altering their code. They're useful for logging, retrying failed steps, or timing function execution in data pipelines.

Solution:

```
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time() - start} seconds")
        return result
    return wrapper
```

Describe the differences between multithreading and multiprocessing in Python, and when to use each in data engineering.

Answer: multiprocessing runs separate processes, ideal for CPU-bound tasks, while multithreading runs threads within one process, better for I/O-bound tasks.

How would you use Python to schedule and orchestrate a data pipeline?

Answer: Use a scheduler like Apache Airflow, which allows scheduling tasks with Python code, monitoring dependencies, and managing task success and failure.

What strategies would you use in Python to avoid code duplication and improve maintainability in a data processing pipeline?

Answer: Modularize code, use functions and classes, apply the DRY (Don't Repeat Yourself) principle, and employ configuration files for parameterization.

How would you handle time zone differences in timestamp data when processing data from multiple geographic regions?

Answer: Use pytz or dateutil libraries to standardize timestamps. Always store data in UTC, then convert as needed.

Solution:

```
from datetime import datetime
import pytz

utc_time = datetime.now(pytz.UTC)
```

Best Practices for Writing Python Code...

- Follow PEP 8 Guidelines
- Use Meaningful Variable and Function Names
- Use List Comprehensions Where Appropriate
- Use try-except Blocks for Error Handling
- Leverage Built-in Functions and Libraries
- Keep Functions Short and Focused
- Use with Statements for File Handling
- Write Tests for Your Code
- Use Virtual Environments
- Document Your Code
- Use Type Annotations
- Avoid Global Variables
- Use F-Strings for Formatting
- Optimize for Performance with Generators
- Keep Your Code DRY (Don't Repeat Yourself)

Python Roadmap for FREE……

https://www.linkedin.com/posts/ajay026_roadmap-comments-python-activity-7102496405306445824-KOQr?utm_source=share&utm_medium=member_desktop

Learn Python from 190+projects here…..

https://www.linkedin.com/posts/ajay026_python-projects-with-source-code-aman-kharwal-activity-7024956726936313856-n0tV?utm_source=share&utm_medium=member_desktop

Python Cheatsheet for Interviews…..

https://www.linkedin.com/posts/ajay026_python-cheat-sheet-activity-7002139681110855680-atQd?utm_source=share&utm_medium=member_desktop

Youtube Channels to Learn Python for FREE…..

https://www.linkedin.com/posts/ajay026_projects-journey-python-activity-7032559987675602944-BmtK?utm_source=share&utm_medium=member_desktop

5 Python Data Analysis Projects to Practice for FREE….

https://www.linkedin.com/posts/ajay026_dataanalytics-python-projects-activity-7010505103870738432-U5y5?utm_source=share&utm_medium=member_desktop

Follow me here on LinkedIn for more information….

https://www.linkedin.com/in/ajay026/